DOCUMENT RESUME

ED 063 776 EM 009 971

AUTHOR	Siklossy, L.; And Others
TITLE	DATADRAW: A Command Language for Manipulating and Displaying Stacks, Queues, Lists and Trees.
INSTITUTION	Texas Univ., Austin. Computation Center.
SPONS AGENCY	National Science Foundation, Washington, D.C.
REPORT NO	TSN-25
PUB DATE	Apr 72
NOTE	47p.
EDRS PRICE	MF-$0.65 HC-$3.29
DESCRIPTORS	*Computer Science; *Data Processing; *Information Processing; *Information Science; *Programing Languages
IDENTIFIERS	CDC 6600; DATADRAW

ABSTRACT

DATADRAW is a command language, written in FORTRAN IV, to manipulate stacks, queues, lists and trees, and to display them on the CDC 252 scope system attached to a CDC 6600. A DATADRAW primer is given, and the algorithms for updating and displaying structures are described. It is noted that DATADRAW was designed to be a simple command language that students could use to familiarize themselves with data structures. (Author/RH)

ED 063776

THE UNIVERSITY OF TEXAS AT AUSTIN

COMPUTATION CENTER
DEPARTMENT OF COMPUTER SCIENCES

EM-00471

DATADRAW: A Command Language for Manipulating and Displaying Stacks, Queues, Lists and Trees*

by

L. Siklóssy, L. Shroyer, and A. Blocher

April 1972 TSN-25

U.S. DEPARTMENT OF HEALTH, EDUCATION & WELFARE
OFFICE OF EDUCATION
THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL OFFICE OF EDUCATION POSITION OR POLICY.

*Work partially supported by NIMH Grant 15769.

THE UNIVERSITY OF TEXAS AT AUSTIN
COMPUTATION CENTER
DEPARTMENT OF COMPUTER SCIENCES

TABLE OF CONTENTS

		Page
Abstract		1
1. Introduction		2
2. A DATADRAW Primer		2
3. Semi-formal Definition of DATADRAW		8
3.1	Identifiers	9
3.2	EXIT	10
3.3	ACTIVE	10
3.4	STATUS	10
3.5	CREATE	11
3.6	KILL	12
3.7	DISPLAY	13
3.8	RENAME	14
3.9	PUT	14
3.10	COPY	14
3.11	DELETE	15
3.12	ADD	16
3.13	INSERT	16-2
3.14	PUSH and POP	16-3
3.15	TRAVERSE	16-3
4. Internal Representations		17
4.1	Lists, Trees and Binary Trees	17
4.2	Stacks and Queues	18
4.3	Storage Management	18
4.4	Other Storage Areas	19
4.5	Note on Storage Efficiency	20
5. Notes on Some Algorithms used in DATADRAW		20
5.1	TREE and BTREE Information Management	20
5.2	DISPLAY of LISTs	21
5.3	DISPLAY of TREEs	23
5.4	DISPLAY of BTREEs	23-1
5.5	RANDOM Structure Generation	23-1
6. Synopsis of DATADRAW Commands		24
7. Error Messages		25
8. Acknowledgment		26
9. References		26
Figures		27

ABSTRACT

DATADRAW is a command language to manipulate stacks, queues, lists and trees, and to display them on the CDC 252 scope system attached to a CDC 6600. A DATADRAW primer is given, and the algorithms for updating and displaying structures are described. DATADRAW is written in FORTRAN IV.

1. INTRODUCTION

DATADRAW is a set of FORTRAN IV programs that permit the user, sitting at the CDC 252 scope terminal attached to the CDC 6600 computer, to create, modify and draw data structures. At present, the permissible structures are stacks, queues, lists and trees. These are some of the most common data structures used in the information sciences.

The development of DATADRAW was motivated by three considerations:

1) to study efficient algorithms for generating, manipulating and displaying data structures.

2) to provide a simple command language that students could use to familiarize themselves with data structures. (The system was used successfully for this purpose in the Fall of 1971).

3) to construct a system that "knows" data structures (in an operative sense) and which could be used as the performance program of a knowledgeable computer tutor [1].

In this report, we give examples of the use of DATADRAW, briefly describe its syntax and semantics, and explain the more interesting algorithms used. Section 2 is a primer on DATADRAW. Section 3 gives a brief, but complete description of the command language. Section 4 explains storage management in DATADRAW. Section 5 describes some of the algorithms used in DATADRAW. Section 6 is a synopsis of DATADRAW commands. Section 7 lists the error messages generated by DATADRAW.

2. A DATADRAW PRIMER

2.1 Entry in DATADRAW

To enter DATADRAW, run the following program for example:

```
ABCD123,John Doe                    (ID card)
PASSWORD CARD
JOB,TM = 30.
RFL,77000.
ASSIGN,FM,FM.
READPF,3195, LGO.
LGO.
7/8/9card.
```

When this program has been executed, a light spot (called the column indicator) will appear on the screen of the scope (if it is well focused). At that point type

CLANG;

The semi-colon serves as a carriage return.

This command (pronounced KLANG) gives you access to the Command LANGuage. We shall run through some manipulations of trees to exemplify some of the capabilities of DATADRAW.

In case of typing errors, the character ≡ is used to backspace, while the character % is used to erase the whole input line. Wherever a space occurs in a DATADRAW command, several spaces can be used. Error messages are erased by a ";".

Perhaps the easiest way to get started is to have DATADRAW create a random tree for you. The command

CREATE TREE T1 RANDOM;

will create a TREE, and will name it T1. Spaces can precede the ;. Executing

STATUS;

will display in the top right corner of the screen all the structures known to DATADRAW at that point. The display will be:

*T1 TREE

indicating that there is a TREE, named T1. The * in front of T1 indicates that T1 is the active structure, i.e. the one presently

looked at by DATADRAW.

To display the tree, enter the command:

DISPLAY;

The result is shown in Figure 1. The above command DISPLAYs the presently active structure on the whole screen. It is equivalent to the command:

DISPLAY T1;

It is also possible to

DISPLAY TOP;

DISPLAY BOTTOM;

DISPLAY TOP RIGHT;

DISPLAY BOTTOM LEFT;

DISPLAY T1 TOP LEFT; etc.

and to give specific coordinates:

DISPLAY 200 BY 300 AT 200 400;

The first two numbers give the width and height of a rectangle in which the figure will be drawn; the last two numbers give the coordinates of the bottom left corner of the rectangle. The whole scope screen is a square 1024 by 1024. The bottom left corner has coordinates 0, 0; the top left corner has coordinates 0, 1024, etc.

If T1 is no longer needed, do:

KILL T1;

which will delete T1 from the STATUS list, and reclaim the memory cells used by T1. After the KILL command, T1 is no longer living, and

DISPLAY T1;

will result in an error.

The command:

KILL EVERYTHING;

will delete all created structures from the system.

The command:

EXIT;

will make you EXIT from DATADRAW to MAIN. To terminate your job, type:

DROP;

after having EXITed from DATADRAW.

After having KILLed T1, let us build some additional structures. We need to erase the screen. A * in column 1 is interpreted as a command to erase the screen:

*;

The erasing * can be used in column 1 of any DATADRAW command, and is executed before that command.

We now build up Figure 2 piece by piece:

	line reference for Comments
*CREATE TREE TR (AAA);	1
DISPLAY TOP LEFT;	2
ADD NODE BBB BELOW AAA;	3
DISPLAY TR TOP RIGHT;	4
IN TR ADD NODE CCC = XXX LEFT BBB;	5
DISPLAY BOTTOM LEFT;	6
ADD NODE DDD = AAA RIGHT CCC;	7
DISPLAY BOTTOM RIGHT;	8

Figure 2 is the picture taken at that point.

A few remarks:

--nodes are specified by their labels, and may or may not have contents. Node AAA has no contents; while DDD has contents AAA. Up to three alphanumeric characters are allowed for names and labels. Alphanumeric labels must begin with an alphabetic character. If longer names are given to labels and contents, only the first three characters are kept. The system will make sure that all labels in <u>one</u> structure are different.

Contents may be the same.

--the TR in line 4 is not necessary (see lines 2,6 and 8). The ACTIVE structure is assumed, if not specified.

--similarly, IN TR (line 5) is not needed. If missing, the ACTIVE structure is assumed.

--line 7 could have been replaced by

ADD NODE DDD = AAA LEFT BBB;

Let us modify the tree. We can change the contents of nodes:

* PUT IN CCC; (The * erases the screen)

PUT 123 in BBB;

We can delete nodes:

DELETE NODE DDD; and

DISPLAY TOP LEFT;

will result in the upper part of Figure 3.

ADD NODE DDD = XYZ RIGHT CCC;

DISPLAY BOTTOM;

will complete Figure 3.

It is possible to input a whole tree given in parenthesized notation. The command at the bottom of Figure 3:

CREATE TREE TO3 (X(Y(Z) W)); followed by:

* DISPLAY LEFT;

results in the left-hand part of Figure 4. Notice that CREATE changed the ACTIVE structure from TR to TO3. A STATUS command would display:

TR TREE

* TO3 TREE

in the top right corner.

It is possible to ADD a TREE to another TREE. We shall make a copy of TREE TR, and call the copy TO2.

IN TR COPY AAA TO TO2;

At this point TO2 becomes active. We can do the ADD:

ADD TREE TO3 BELOW CCC;

DISPLAY TO2 RIGHT;

Figure 4 is a picture taken at this point.

It is possible to RENAME structures or nodes. Figure 5 is the result of:

*ACTIVE TR;

DISPLAY TOP LEFT;

RENAME NODE Y AS GGG;

PUT AAA IN W;

DISPLAY BOTTOM;

Figure 6 shows how a sub-tree is RENAMEd. The tree on the LEFT is called X24. To obtain the tree on the RIGHT, when TR is active, we can do:

IN X24 RENAME TREE B12 AS X30;

ACTIVE X24;

DISPLAY RIGHT;

The new label must be a letter followed by two digits. Labels are obtained by adding 1 (modulo 100) to the last two digits, and prefixing with the letter.

To change the name of the TREE X24 to LBJ, execute the command:

RENAME STRUCTURE X24 AS LBJ;

It is possible to copy a sub-tree. In Figure 7, if the TREE at the LEFT is FDR, we obtain the rest of the figure by executing:

ACTIVE FDR;

COPY SO3 TO XXX;

DISPLAY XXX TOP RIGHT;

After the COPY, XXX is active, hence we need not specify XXX for DISPLAY.

Figure 8 terminates our examples in this DATADRAW primer. After the commands:

ACTIVE T3;

DISPLAY;

the command:

TRAVERSE PRE ;

was executed. The TREE T3 is traversed in PREorder. Successive semi-colons, i.e. ;'s, result in numbers pointing to the successive nodes encountered during PREorder traversal.

In DATADRAW, binary trees (or BTREEs), LISTs and TREEs can be traversed in PREorder, POSTorder, ENDorder or LEVEL order.

A DISPLAY command must have been previously executed with no * commands executed between the DISPLAY and the TRAVERSE.

3. Semi-Formal Definition of DATADRAW

In this section we shall give the syntax and semantics of the commands in DATADRAW. The treatment will not be completely formal. A mixture of BNF notation and linguistic conventions will be used to describe the syntax of DATADRAW. Square brackets indicate that a choice must be made among several statements placed one below the other in the square brackets. Curly brackets indicate that a choice can be made among several statements placed one below the other in the curly brackets, but the empty choice is also allowed.

DATADRAW commands are placed on a single line of input, which terminates with a ";". The command is executed after the ";" has been read. The backspace character is: "≡"; while the line delete character is: "%".

3.1 Identifiers

3.1.1 Command words.

The following is an alphabetic listing of command words:

ABOVE	END	PUSH
ACTIVE	EVERYTHING	PUT
ADD	EXIT	QUEUE
AS	IN	RANDOM
AT	INSERT	RENAME
BELOW	KILL	RIGHT
BOTTOM	LEFT	STACK
BTREE	LEVEL	STATUS
BY	LIST	STRUCTURE
COPY	NODE	TO
CREATE	POP	TOP
DELETE	POST	TRAVERSE
DISPLAY	PRE	TREE

3.1.2 Constants

type : := BTREE | LIST | QUEUE | STACK | TREE

3.1.3 Variables

"name" "cont" and "label" must be three character alphanumeric identifiers, unless otherwise indicated. Alphanumeric identifiers must start with an alphabetic character, unless they are purely numeric. For ease of understanding, "name" will be the name of a structure, "label" the label of a node, and "cont" the contents of a node. Indices will distinguish possibly different variables of a similar character. "num" must be an unsigned integer.

3.1.4 Special Characters

A * as the first character of a command line is executed before the command, and erases the screen.

3.2 EXIT

3.2.1 Syntax

EXIT;

3.2.2 Semantics

The program exits from DATADRAW into MAIN, the program that activates the scope. The command:

DROP;

will terminate the program.

3.3 ACTIVE

3.3.1 Syntax

ACTIVE name;

3.3.2 Semantics

At any time, at most one structure is ACTIVE. The command ACTIVE makes the structure "name" active. Certain commands, such as CREATE, change the ACTIVE structure.

3.4 STATUS

3.4.1 Syntax

STATUS;

3.4.2 Semantics

The name and type of each structure in the system are displayed in the top right corner of the screen. A * is placed in front of the ACTIVE structure.

3.5 CREATE

3.5.1 Syntax

CREATE <u>type</u> {name} [<list notation> | RANDOM {parameters}]

3.5.2 Semantics

If name is missing, the label of the first node of the structure is taken as the name of the structure.

The conventions of the list notation are as follows:

--for TREES: the labels of the sons of a node are at the top level of a list following the father. Examples:

(A = B (C = B (D E G) F = B)) is a tree with root A, contents B. The two sons of A are C and F, both with contents B, and C has three sons labelled D, E and G. The list notation for the tree in Figure 4, RIGHT, would be: (AAA(CCC(X(Y(Z)W)) DDD = XYZ BBB = 123)).

--for LISTs: sublists of a node are parenthesized and follow the node. Example: the list at BOTTOM RIGHT of Figure 9 could be input as (AAA (BBB = N2 DDD = N2) CCC = N3).

-- for BTREEs (<u>B</u>inary <u>TREE</u>s): the conventions are the same as for TREEs with two exceptions; (1) a maximum of two labels may appear at any level of the list notation, and (2) the label following an open parenthesis is assumed to be a left son. If no left son exists, then the character "≠" is used in its place to force the label of the right son to be recognized. The list notation for the BTREE at the bottom left of Figure 11 would be: (A (B = BB(≠D(E))C)).

--for STACKS: a simple list is considered a stack, the right of the list is the top of the stack.

--for QUEUEs: a simple list is considered a stack, the left of the list is the front of the queue.

There are five optional parameters IPL IPR IPLR NUM IALPHA in RANDOM. Their range is as follows:

$0 \leq$ IPL, IPR, IPLR ≤ 100; IPL + IPR + IPLR ≤ 100; $1 \leq$ NUM ≤ 50; $1 \leq$ IALPHA ≤ 26.

The parameterless call to RANDOM:

CREATE type {name} RANDOM;

is equivalent to the call:

CREATE type {name} RANDOM IPL1 IPR1 IPLR1 NUM1 IALPHA1;

where:

IPLR1 = RAND(30,70); TEMP = RAND(10,30); IPL1 = RAND(0,100-IPLR1-TEMP); IPR1 = 100-IPLR1-IPL1-TEMP; NUM1 = RAND(2,30); IALPHA = RAND(1,26); and RAND(N,M) generates a random number between N and M. The meaning of these parameters is tied to the random structure generator, which is described in section 4.

Random stacks and queues are equivalent to generating a random list of length N, where N = RAND(2,30).

CREATE will generate an error diagnostic if the "name" given is already in the STATUS table, independently of the type of the structure. CREATE also checks for well-formedness, for example, for multiple identical labels.

After a successful CREATE, the new name is entered in the STATUS list, and the new structure becomes ACTIVE.

3.6 KILL

3.6.1 Syntax

KILL [name | EVERYTHING] ;

3.6.2 Semantics

KILL name; removes the structure "name" from the STATUS list and regains all the storage space used up by the structure. If name was ACTIVE, no structure will be ACTIVE after the KILL.

KILL EVERYTHING; removes all structures from the STATUS list and cleans up memory. Since no structures are left, none can be ACTIVE.

3.7 DISPLAY

3.7.1 Syntax

DISPLAY { name } { { TOP / BOTTOM } { LEFT / RIGHT } / num$_1$ BY num$_2$ AT num$_3$ num$_4$ }

3.7.2 Semantics

DISPLAY will draw the structure name on the screen, independently of what is already on the screen (so overwriting is possible). If name is missing, the ACTIVE structure is assumed.

The optional numerical parameters specify a rectangle num$_1$ BY num$_2$ with lower left corner at the point with coordinates (num$_3$ num$_4$). The other specifications are abbreviations for specific values of the numerical parameters.

Abbreviation	Num$_1$ (width)	Num$_2$ (height)	Num$_3$ (x)	Num$_4$ (y)
BOTTOM	990	450	33	123
BOTTOM LEFT	495	450	33	123
BOTTOM RIGHT	495	450	528	123
LEFT	495	900	33	123
RIGHT	495	900	528	123
TOP	990	450	33	573
TOP LEFT	495	450	33	573
TOP RIGHT	495	450	528	573
missing	990	900	33	123

The entire screen is: 1024 BY 1024 AT 0 0.

The DISPLAY routines are explained in section 5.

3.8 RENAME

3.8.1 Syntax

$\left\{ \text{IN name}_1 \right\}$ RENAME $\left[\begin{matrix} \text{node} \\ \underline{\text{type}} \end{matrix} \right]$ label_1 AS label_2;

RENAME STRUCTURE name_1 AS name_2;

3.8.2 Semantics

If $\left\{ \text{IN name}_1 \right\}$ is missing, the name of the ACTIVE structure is assumed.

In the first version of RENAME, if the second option is NODE, a search is made for a label equal to label_1 in the structure name_1. If found, this label is replaced by label_2. If label_1 is not found, a message is given. If the second option is type, label_2 must be a letter followed by two digits, as for example C34. The structure (i.e. subtree or sublist) rooted at label_1 is traversed in PREorder and labels changed to C34, C35, etc. C01 follows C99. In the second version of RENAME, only the name_1 of the data structure is changed to name_2. This is the only meaningful RENAME for stacks and queues.

3.9 PUT

3.9.1 Syntax

$\left\{ \text{IN name} \right\}$ PUT $\left\{ \text{cont} \right\}$ IN label;

3.9.2 Semantics

If $\left\{ \text{IN name} \right\}$ is missing, the name of the ACTIVE structure is assumed.

A search is made for a node label equal to "label". If "label" is found, its contents are replaced by that given in the command. If label is not found, a message is given.

3.10 COPY

3.10.1 Syntax

$\left\{ \text{IN name}_1 \right\}$ COPY label TO name_2;

3.10.2 Semantics

If {IN name$_1$} is missing, the name of the ACTIVE structure is assumed. A search is made for a node label equal to label in name$_1$ of the ACTIVE structure. If label is found, the substructure rooted at label is copied to a new structure called name$_2$. Name$_2$ will be active after the COPY. If label is not found, a message is given. If name$_2$ is not a unique name, a message is given.

3.11 DELETE

3.11.1 For a queue.

{IN name} DELETE;

If {IN name} is missing the ACTIVE structure is assumed. The front of the queue is DELETEd.

3.11.2 For a TREE or BTREE.

{IN name} DELETE {BTREE / TREE / NODE} label;

If {IN name} is missing, the ACTIVE structure is assumed.
If the second option is missing, NODE is assumed.
The substructure rooted at "label" in "name" is deleted. If label is not found, a message is given.

3.11.3 For a LIST.

{IN name} DELETE {LIST / NODE} label;

If {IN name} is missing, the ACTIVE structure is assumed.

If the second option is missing, NODE is assumed. When NODE is assumed or specified, three cases can occur:

a) if "label" is linked by a right link from some other node, "label" and everything linked under it is deleted. (The left brother of "label" is linked to the right brother of "label" (if any) by a right link).

b) If label is linked by a down link from some other node, "label" and everything linked to the right of it is deleted. The (up) father of "label" is linked to the (down) son of "label" (if any) by a down link.

c) If label is the root node, the complete structure is deleted.

If LIST is used as the second option, the complete substructure rooted at "label", in "name", is deleted. If label is not found, an error message is given.

3.12 ADD

3.12.1 For a queue.

{IN name} ADD cont;

If {IN name} is missing, the ACTIVE structure is assumed.

"cont" is ADDed to the rear of the queue.

3.12.2 For a BTREE

$$\left\{\text{IN name}_1\right\}\ \text{ADD}\ \left[\begin{matrix}\text{BTREE name}_2\\ \text{NODE label}_1\ \{=\text{cont}\}\end{matrix}\right]\ \left[\begin{matrix}\text{LEFT}\\ \text{RIGHT}\end{matrix}\right]\ \text{label}_2;$$

If IN $name_1$ is missing, the ACTIVE structure is assumed.

If the "BTREE $name_2$" option is chosen, an existing BTREE named $name_2$ will be added as the left or right substructure of the node $label_2$. If $label_2$ is not found an error message is given.

If the "$label_1$" option is chosen, a single node named $label_1$ is created as a left or right son of $label_2$.

3.12.3 For a TREE

$$\left\{\text{IN name}_1\right\}\ \text{ADD}\ \left[\begin{matrix}\text{TREE name}_2\\ \text{NODE label}_1\ \{=\text{cont}\}\end{matrix}\right]\ \left[\begin{matrix}\text{BELOW}\\ \text{LEFT}\\ \text{RIGHT}\end{matrix}\right]\ \text{label}_2;$$

If $\left\{\text{IN name}_1\right\}$ is missing, the ACTIVE structure is assumed.

If the "TREE $name_2$" option is chosen, an existing TREE named $name_2$ will be added according to the following rules.

If the BELOW option is chosen, the structure is added as the

rightmost son of $label_2$. If the LEFT (or RIGHT) option is chosen, the structure is added to the father of $label_2$ and to the immediate left (or right) of $label_2$.

If the "$label_1$" option is chosen, a single node named $label_1$ is created and added according to the above conventions for BELOW, LEFT or RIGHT.

3.12.4 For a LIST

$$\left\{\text{IN name}\right\}\ \text{ADD}\ \left\{\begin{matrix}\text{LIST}\\ \text{NODE}\end{matrix}\right\}\ label_1\ \left\{=\text{cont}\right\}\ \left[\begin{matrix}\text{ABOVE}\\ \text{BELOW}\\ \text{LEFT}\\ \text{RIGHT}\end{matrix}\right]\ label_2;$$

If $\left\{\text{In name}\right\}$ is missing, the ACTIVE structure is assumed. If the second option is missing, NODE is assumed.

When NODE is assumed or specified, several cases can occur:

a) $label_2$ has no son (or right brother) and BELOW (or RIGHT) are chosen. If the BELOW option is chosen, and the node labelled $label_2$ has no son (i.e. has no down link), the node $label_1\left\{=\text{cont}\right\}$ is added as the son of $label_2$. (Similarly, if the RIGHT option is chosen, and $label_2$ has no right brother (i.e. has no right link), the node $label_1\left\{=\text{cont}\right\}$ is added as the right brother of $label_2$.)

b) $label_2$ already has a son (or right brother) and BELOW (or RIGHT) are chosen. If the BELOW option is chosen and $label_2$ already has a son (i.e. a down link) then this son (with all its descendants and right brothers) is deleted, and $label_1\left\{=\text{cont}\right\}$ becomes the new right brother of $label_2$. Note that in both of these cases, a single node may replace an entire structure.

c) If the LEFT option is chosen, ADDition can only occur if $label_2$ has no left brother. If $label_2$ is not the root of the list,

label$_1$ becomes the new leftmost son of the father of label$_2$, while label$_2$ becomes the right brother of label$_1$. If label$_2$ is the root, label$_1$ becomes the new root and label$_2$ becomes the right brother of label$_1$.

Similarly, if the ABOVE option is chosen, ADDition can only occur if there is no down link to label$_2$. This implies that, unless label$_2$ is the root, it has a left brother. Label$_1$ becomes the new right brother of this left brother, while label$_2$ becomes the son of label$_1$. If label$_2$ is the root, label$_1$ becomes the new root, and label$_2$ becomes its son.

The above conventions were chosen for their intuitive appeal. They are not standard. In fact, no standard conventions seem to exist!

When LIST is used as the second option, the ABOVE and LEFT options have no meaning and result in an error message. In the case of the BELOW and RIGHT options, the operations described in a) and b) above occur with the exception that the entire list structure named label$_1$ is added instead of a single node. Obviously, label$_1$ may not have contents in this case.

3.13 INSERT

INSERT is only meaningful as an operation on LISTs.

3.13.1 Syntax

$$\left\{\text{IN name}\right\}\ \text{INSERT}\ \left\{\text{NODE}\right\}\ \text{label}_1\ \left\{\text{=cont}\right\}\ \left[\begin{matrix}\text{ABOVE}\\ \text{BELOW}\\ \text{LEFT}\\ \text{RIGHT}\end{matrix}\right]\ \text{label}_2;$$

3.13.2 Semantics

If IN name is missing, the ACTIVE structure is assumed. The second option, NODE, is always assumed, whether present or not.

The single node label$_1$ $\left\{\text{=cont}\right\}$ will be INSERTed ABOVE,BELOW,LEFT or RIGHT of the node label$_2$ if the appropriate preconditions are

satisfied:

a) ABOVE (or LEFT). There must be a down (or right) link to $label_2$ from some node $label_3$. $Label_3$ is linked down (or right) to $label_1$, which is linked down (or right) to $label_2$.

b) BELOW (or RIGHT). There must be a down (or right) link from $label_2$ to some node $label_3$. $Label_2$ is linked down (or right) to $label_1$, which is linked down (or right) to $label_3$.

3.14 PUSH and POP

PUSH and POP are addition and deletion operations on STACKs.

3.14.1 Syntax

```
{IN name} [ POP       ] ;
          [ PUSH cont ]
```

3.14.2 Semantics

If IN name is missing, the ACTIVE structure is assumed. (It must be a STACK.)

POP deletes the top element of the STACK. An empty STACK may not be POPped.

PUSH adds cont as the contents of the new top of the STACK. The STACK becomes one cell longer.

3.15 TRAVERSE

TRAVERSE can be used on TREEs, BTREEs and LISTs.

3.15.1 Syntax

```
TRAVERSE [ END   ] ;
         [ LEVEL ]
         [ POST  ]
         [ PRE   ]
```

3.15.2 Semantics

The last structure displayed is TRAVERSEd in endorder, level order, postorder or preorder. Successive pressing of the character ';' will result in arrow--numbered 1,2,3,...-- to point to the nodes in the order in which they are visited by the traversal.

TRAVERSE makes use of a table containing the absolute screen coordinates of the structure that has been DISPLAYed last. Figure 8 shows the PREorder traversal of a RANDOM tree.

4. Internal Representations.

A common M x 8 array is used by all the data structures.

4.1 Lists, Trees and Binary Trees.

LISTs, TREEs and BTREEs make the same use of the first four columns of the storage array.

Column	1	2	3	4
	name of node	content	Link 1	Link 2
LIST	"	"	DOWN	RIGHT
TREE	"	"	SUB	PRED
BTREE	"	"	LEFT	RIGHT

In a TREE, the SUB points to the left-most son of a node, while PRED points to the next brother to the right. The other links should be obvious.

4.1.1 TREEs and BTREEs

Columns 5 to 8 have similar meanings for TREEs and BTREEs:

Column	5	6	7	8
TREE	right-most son	father	weight	level
BTREE	(not used)	father	weight	level

In both cases, the level of a node is one plus the length of the path leading to the root of the tree. The root of the tree has level 1, its sons level 2, etc.

Weights are used in the display routines.

--TREES. The weight of a node is obtained recursively as follows:

a) the weight of a terminal node is 1.

b) the weight of a non-terminal node is the sum of the weights of its sons.

--BTREEs. The weight of a node is an integer obtained recursively as follows:

a) the weight of a terminal node is 1.

b) the weight of a non-terminal node is:

weight of the heaviest son + max(weight of lightest son , 1/4. weight of heaviest son),

where the result of the multiplication is truncated.

The weight of a non-existent son is 0.

4.1.2 LISTs.

Columns 5 to 8 for LISTs give information to the LIST display routine.

Column	5	6	7	8
LIST	horizontal D displacement	horizontal L displacement	vertical D displacement	vertical L displacement

Columns 5 to 8 for TREEs and BTREEs are updated as nodes or structures are ADDed, INSERTed or DELETEd from the structure. For LISTs, columns 5 to 8 are filled in only prior to the DISPLAY of the LIST.

4.2 STACKs and QUEUEs.

The same M x 8 array storage is used for STACKs and QUEUEs as for LISTs, TREEs and BTREEs. Of the eight columns, seven are used for storing contents of the structure, while the eighth column is used as a link to the next index of the array in which the structure continues. Special contents (-1) are used to indicate the end of a structure in a particular row of the storage array.

4.3 Storage Management

Originally, the storage array is linked through the 3rd column to form available space. DATADRAW commands to CREATE, COPY, ADD and INSERT obtain array elements from available space. Commands that DELETE or KILL return released array indices to available space. There is no garbage collector since all garbage collection is done locally and immediately upon

release of space.

4.4 Other Storage Areas

Besides the main storage array described above, there are smaller storage areas in DATADRAW:

a) The STATUS list.

b) A storage area for the absolute scope coordinates needed to display a structure. They depend on the region of the scope on which the structure will be displayed. The absolute coordinates of only one structure are kept at any one time: it is the last structure which was displayed.

c) An error array which contains all the error diagnostics (see section 7).

d) An array used by the scan routine that reads DATADRAW commands. The contents of this array are interpreted in context, and executed by the DATADRAW monitor.

e) A working stack is used to traverse structures.

f) A hash table is used when a structure is ADDed to another structure, to check for duplicate node labels. The hashing technique assures that if a structure with M nodes is ADDed to a structure with N nodes, then approximately M + N comparisons need to be made, instead of the approximately M x N if every node of one structure is checked against every node of the other structure.

The buckets of the hash-table are linked lists formed from elements of the main storage area. Seven columns are used for storage, and the eighth column for the list linkage in case more than seven elements hash into the same bucket.

g) Several storage areas are needed for interaction with the scope hardware.

4.5 Note on Storage Efficiency.

Since the words of the CDC 6600 have 60 bits, an estimated saving of 75% of the memory used by the structures could be achieved by packing name, contents and links. This saving would result in somewhat increased computing time, taken up by packing and unpacking operations. A storage array having a row length of 500 has been used, and was amply sufficient, so that memory savings were not required.

5. Notes on some Algorithms used in DATADRAW

In this section we describe some of the more interesting algorithms used in DATADRAW. We have mentioned the hashing technique used to check for conflicts among labels of two structures when one of these is added to the other. The traversal routines we shall mention can be found in Knuth [2]. The management of available space is standard.

5.1 TREE and BTREE Information Management.

As TREEs and BTREEs are changed, the weight and level information is updated, as well as the linkage information, of course. It is interesting to notice that, as a node is added, only the weights of the ancestors need to be modified. The process is simple:

For TREEs:

--if a node is added BELOW a terminal node, no updating is needed.

--if a node is added LEFT or RIGHT, an increment of 1 for the weights of all the ancestors is made.

--if a TREE with a root of weight M is added BELOW terminal node L, updating occurs only if $M > 1$, in which case $(M - 1)$ is added to the

weights of L and its ancestors.

--if a TREE with a root of weight M is added LEFT or RIGHT of node L, all the ancestors of L have their weights increased by M.

For BTREEs, the updating operations are similar, except that the updating might not rise to the root. This can be seen as follows: the formula used for updating includes a max. Therefore, the increase of the weight of a son may not affect the weight of a father if the weight of the other son is sufficiently large.

Originally, weights were computed before display by traversing a BTREE in endorder.

If a subTREE rooted at L is deleted from a TREE, no updating is necessary if the weight of the father of L was 1. Otherwise, all the ancestors of L have their weights decreased by (weight of L) -1. For BTREEs, if a subBTREE is deleted, updating may not rise to the root, again because of the way in which weights are calculated.

5.2 DISPLAY of LISTs.

It is easiest to explain the LIST DISPLAY routine by considering an example: take the figure at BOTTOM RIGHT of Figure 9. The LIST is traversed in PREorder. First, node AAA is encountered. Parametric coordinates for the lower left corner of the box labelled AAA are determined. The coordinates are relative to the top left corner of the area in which the structure will be displayed. The horizontal displacement -x- is counted from left to right; the vertical displacement -y- is counted from top to bottom. The size of the box is 2D by D, while the vertical and smallest horizontal separations between boxes are L. Hence the coordinates of box AAA are (0,D). The traversal moves to box BBB, which has coordinates (0,2D + L). No additional DOWN links are encountered and a RIGHT link is taken to node DDD. The coordinates of

box DDD are (2D +L, 2D + L). Since node DDD is terminal, the traversal climbs back to node CCC. At this point, the horizontal coordinate of the rightmost box encountered in the LIST is kept: it is (2D + L). The horizontal coordinate of node CCC will be 2D + L to the right of that: 2D for the size of the box (here DDD) and L for the horizontal displacement. Hence the coordinates of CCC are (4D + 2L, D). In this way, we are assured that any subLIST rooted at CCC will not be displayed over some previous part of the structure.

After the coordinates of the boxes are obtained in terms of D and L, actual values of D and L are calculated. If the horizontal size of the LIST is mD + m'L, and its vertical size is nD + n'L, and if the area in which it is to be displayed is XSIZE x YSIZE, then two methods could be used to calculate the values of D and L:

a) Solve the two simultaneous equations:

$$mD + m'L = XSIZE$$

$$nD + n'L = YSIZE$$

for the unknowns D and L.

b) Assume a fixed ratio L = kD, and select

$$D = \min(XSIZE/(km + m'), YSIZE/(kn + n')).$$

For esthetics, we let D:= min (D,DMAX) and L:= min(L,LMAX) so that enormous boxes and/or links are not drawn.* The boxes must be sufficiently large so that contents can be read, hence D $\geq$ DMIN, and the links sufficiently long, L $\geq$ LMIN, so that the boxes are not too crowded together. If the structure is too big, the minimum sizes of boxes or links do not allow the structure to be displayed, and an error message results. A larger scope area should be given, or parts of the structure DELETEd.+

---

* TOP LEFT in Figure 2 is a case where the size of the box was limited.

\+ We are implementing a partial DISPLAY routine which will display as much of the routine as there is room in the prescribed scope area, and give information on the parts that could not be displayed.

Subject to the restrictions on maximum and minimum sizes of boxes and links, method a) will fill the whole display area allotted to the screen; while method b) will fill either the vertical or the horizontal coordinates, but seldom both. Methodb) has been used for LISTs, with k = 2, while a variant of method a) has been used for TREEs and BTREEs.

After actual values of D and L have been calculated, absolute scope coordinates are determined and stored in a display table. Scope commands are initiated to draw the structure. The absolute coordinate table is used by the TRAVERSE commands.

It should be noted that the parametric coordinates are determined by a single traversal of the structure, which is a minimum. Moreover, if we wish to display the structure in different areas of the screen, the parametric coordinates need not be recalculated: only the absolute coordinates must be obtained.

5.3 DISPLAY of TREEs

The tree display routine makes use of the weights associated with each node of the tree. The weights (see section 4.1.1) are defined such that each terminal node has a weight of one and all other nodes have weights equal to the sum of the weights of their sons. The weights are used by the display routine to determine the relative horizontal spacing of nodes. The relative vertical spacing of nodes is determined by dividing the display zone height by the number of levels within the tree (but limiting this value to some maximum to avoid very long branches). The display zone height, width and location on the screen are input parameters for the display routine.

The relative positions of nodes within the display zone are determined in the following manner:

(1) The root node is displayed at the top center of the display zone. The left boundary is set to the left boundary of the display zone.

(2) The tree is traversed in preorder with the use of a stack.

(3) Each node is displayed, as it is visited, at the current vertical and horizontal position. The vertical position is decreased each time a sub link is followed and reset from the stack each time the stack is popped. The horizontal position is calculated as the current left boundary plus one-half the product of the father's width and the ratio of the weight of the son to the weight of the father. The current father's width is reset each time a sub link is followed. The current left boundary and the father's width are reset from the stack when it is popped.

5.4 DISPLAY of BTREEs

The binary tree display is done by the same routine as the tree display and in the same manner with the exception of the handling of the weights. The weight of a node of a BTREE was given in section 4.1.1. From the formula giving the weight, it is seen that light or non-existent nodes still contribute to the weight of their father, thereby insuring that all LEFT and RIGHT links will be drawn slanting left and right from the vertical.

5.5 RANDOM Structure Generation.

Building a RANDOM stack or queue is trivial. Hence, we restrict our discussion to the single algorithm that builds RANDOM trees, btrees and lists. The RANDOM command uses five parameters IPL IPR IPLR NUM and IALPHA. The range of these parameters is given in section 3.5.2. These parameters are optional in the call to RANDOM and, if missing, are replaced by default values calculated randomly by DATADRAW (see section 3.5.2).

The meanings of the parameters are:

IPL: probability of building a Link1.

IPR: probability of building a Link2.

IPLR: probability of building both Link1 and Link2.

NUM: number of nodes in the structure.

IALPHA: numeric value of the alphabetic character used for labelling the structure. (If IALPHA equals 2, the nodes will be labelled B01, B02, ..., B99.)

The probabilities are integers in the closed interval (0,100). We then calculate:

P1 = (100 - IPLR - IPL - IPR) * 0.01

P2 = P1 + IPL * 0.01

P3 = P2 + IPR * 0.01

The links and nodes are generated by the following algorithm:

1. A node is generated and becomes the current node. NCTR:=1;

2. IF NCTR $\geq$ NUM, THEN EXIT;

3. A real random number P is generated, with $0.0 \leq P \leq 1.0$;

4. IF $P \leq$ P1, no links are generated, the current node is placed in the array of terminal nodes. Go to 9.

5. If $P1 < P \leq P2$, a left link is generated from the current node, and a new node is attached to the link. The new node is placed in the array of the terminal nodes. NCTR:=NCTR+1. Go to 9.

6. If $P2 < P \leq P3$, a right link is generated from the current node, and a new node is attached to the link. The new node is placed in the array of terminal nodes. NCTR:=NCTR+1. Go to 9.

7. If (NCTR + 2) > NUM, go to 10.

8. Both left and right links are generated from the current node, and new nodes attached to the links. The new nodes are placed in the array of terminal nodes. NCTR := NCTR + 1.

9. A node is randomly removed from the array of terminal nodes. This node becomes the current node. Go to 2.

10. IF P3 < 0.2 THEN EXIT ELSE go to 3.

It is seen that nodes are added somewhat randomly around the structure, until the structure reaches a predetermined number of nodes.

Note that this unique random structure builder is used for TREEs, BTREEs and LISTs. Section 4.1 shows how Link1 and Link2 are interpreted for each of the three structures. (There is only one difference: no Link2 is built from the root of a TREE.)

6. Synopsis of DATADRAW commands.

```
ACTIVE name;
{IN name}  [ADD]     {BTREE}          label1 { =cont }  [ABOVE]      label2;
           [INSERT]  {LIST }                            [BELOW]
                     {NODE }                            [LEFT ]
                     {TREE }                            [RIGHT]

{IN name1}  COPY label1  TO name;

CREATE type {name}  {<list notation>                                  }
                    {RANDOM {IPL IPR IPLR NUM IALPHA}                 } ;

{IN name}  DELETE  {BTREE}
                   {LIST }
                   {NODE }
                   {TREE }   label1;

DISPLAY  {name}  {  {TOP   }  {LEFT }                                 }
                 {  {BOTTOM}  {RIGHT}                                 }
                 {  num1 BY num2 AT num3 num4                         }  ;

EXIT;

{IN name}  INSERT  {NODE}  label1 { =cont }  [ABOVE]      label2;
                                             [BELOW]
                                             [LEFT ]
                                             [RIGHT]

KILL  [name      ] ;
      [EVERYTHING]

{IN name}  [POP      ] ;
           [PUSH cont]

{IN name}  PUT cont IN label1;

{IN name1}  RENAME  [NODE]  label1 AS label2 ;
            RENAME  [type]
            RENAME  STRUCTURE name2  AS name3 ;

STATUS;
TRAVERSE  [END  ]
          [LEVEL]  ;
          [POST ]
          [PRE  ]
```

type : : = BTREE | LIST | QUEUE | STACK | TREE

Convention:
A * in column 1 of a command (possibly empty) erases the screen.

As given in this synopsis, some of the commands are semantically inacceptable.

7. Error Messages

The following are the error messages encountered in DATADRAW. Most of them are self-explanatory.

NO STRUCTURES EXIST

NO ACTIVE STRUCTURES

STRUCTURE NAME ALREADY EXISTS

NO ROOM IN DIRECTORY FOR STRUCTURE

The status table can contain at most 20 names. KILL some structures.

NO ROOM IN STORAGE FOR STRUCTURE

Linked storage space is full.

QUALIFIER INCORRECT

IN name incorrectly used.

ERROR IN SYNTAX

UNRECOGNIZED COMMAND

DISPLAY COORDINATES OUT OF BOUNDS

The rectangle you gave is not inside the screen.

STRUCTURE DOES NOT EXIST

LEFT NODE ALREADY EXISTS
RIGHT NODE ALREADY EXISTS
} Used for binary trees.

IMPOSSIBLE

Catch all.

STRUCTURE TOO LARGE FOR DISPLAY AREA

NODE ALREADY EXISTS

IMPOSSIBLE--DUPLICATE NODE NAMES

In trying to add one structure to another, it was found that some of the node names were common.

REQUESTED OPERATION INVALID

SYSTEM ERROR

This is a challenging error.

WORKING STACK OVERFLOWED

You must have created an exceedingly long list somewhere.

8. Acknowledgment

J. Peach helped us in the early stages.

9. References

1. Siklóssy, L. Computer Tutors that know what they teach. Proc. Fall Joint Computer Conf., 251-255, 1970.

2. Knuth, D. The Art of Computer Programming, Vol. 1. Addison-Wesley, Reading, MA. 1968.

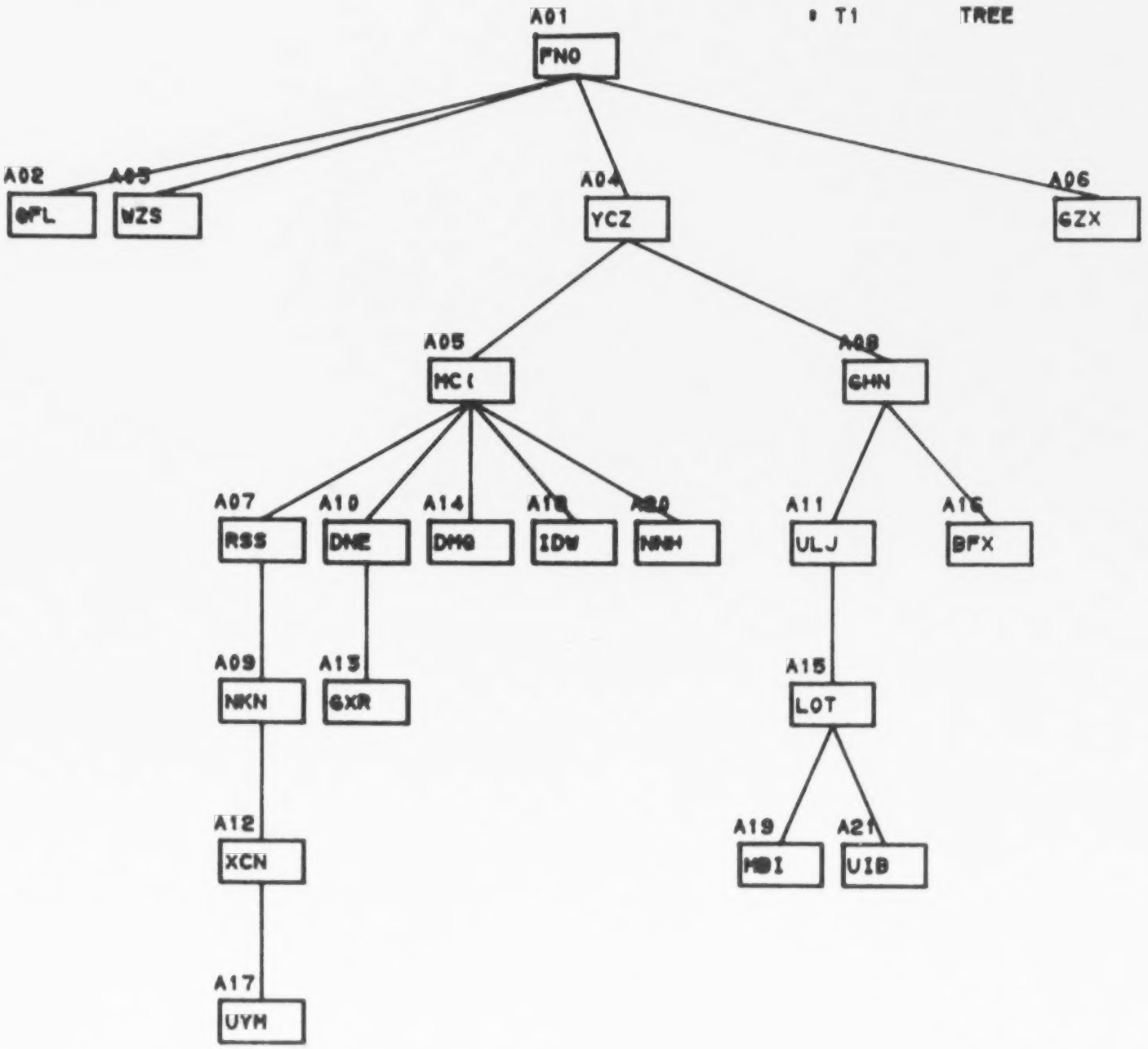
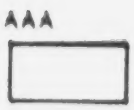
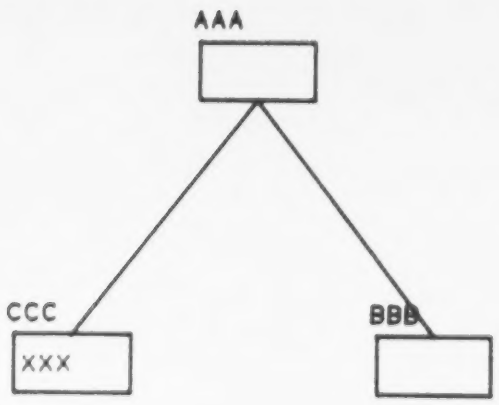
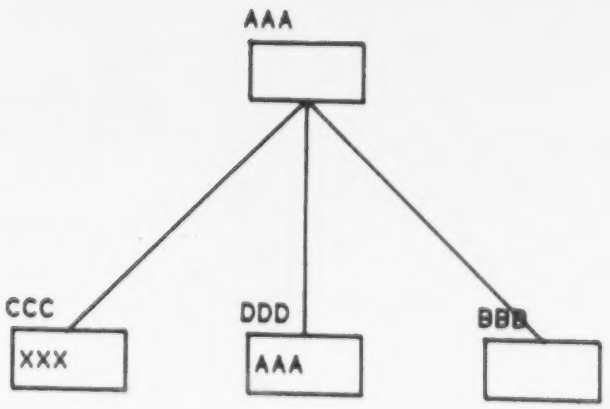

A01
FNO
T1
TREE
A02
OFL
A03
WZS
A04
YCZ
A06
GZX
A05
MC(
A08
GHN
A07
RSS
A10
DNE
A14
DMG
IDW
NNH
A11
ULJ
BFX
A09
NKN
A13
GXR
A15
LOT
A12
XCN
A19
MBI
A21
UIB
A17
UYM

Figure 1

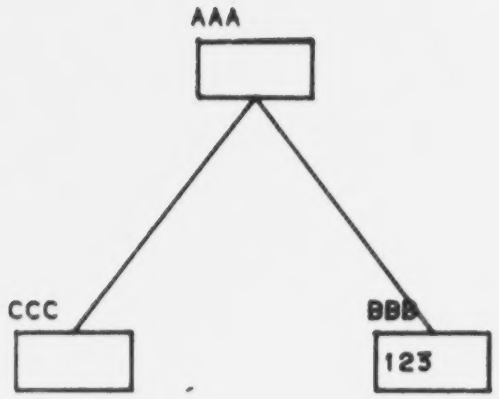


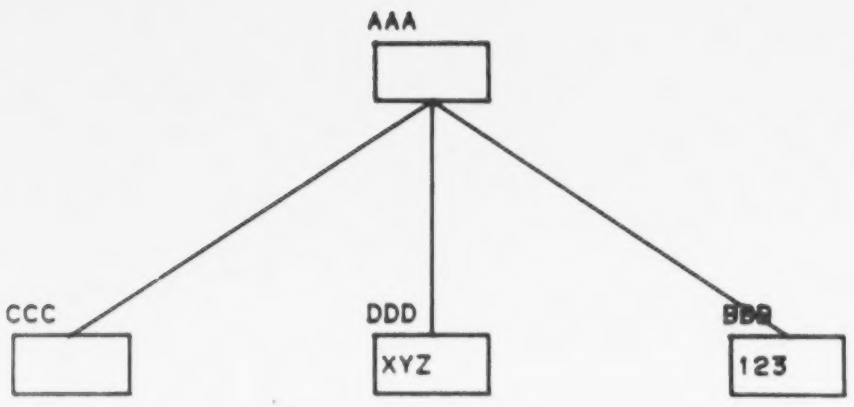





```
DISPLAY BOTTOM RIGHT;
```

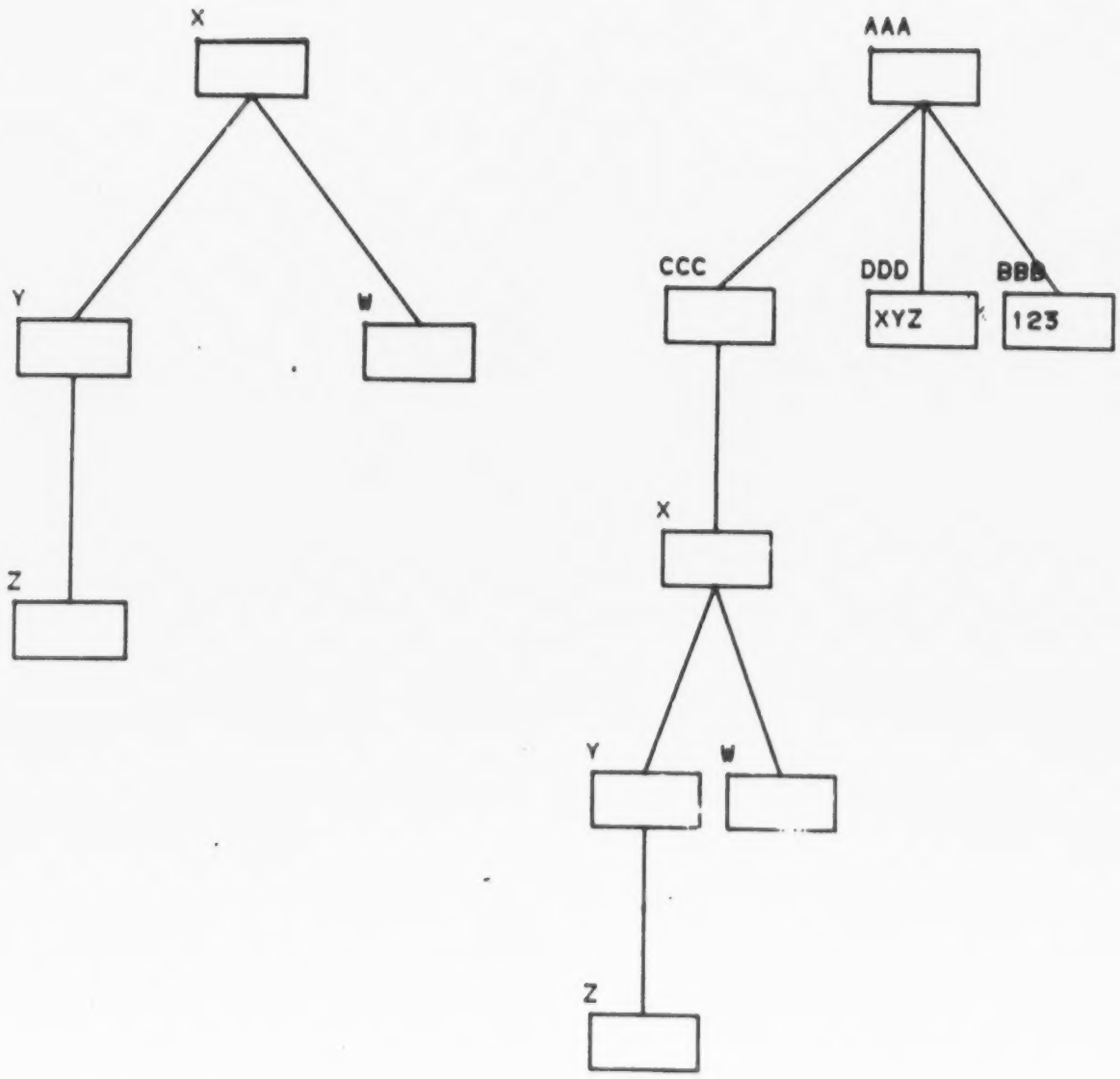
Figure 2





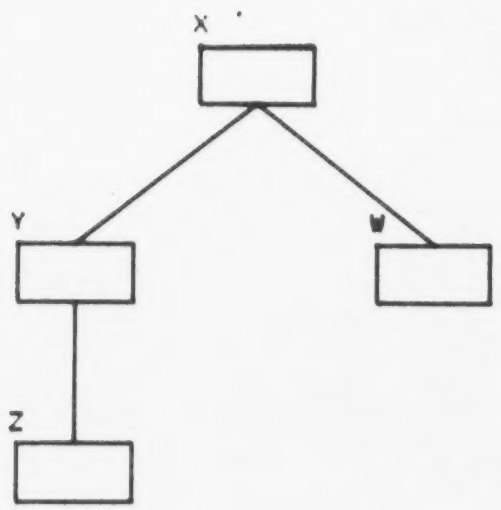

```
CREATE TREE T03 (X(Y(Z) W))
```

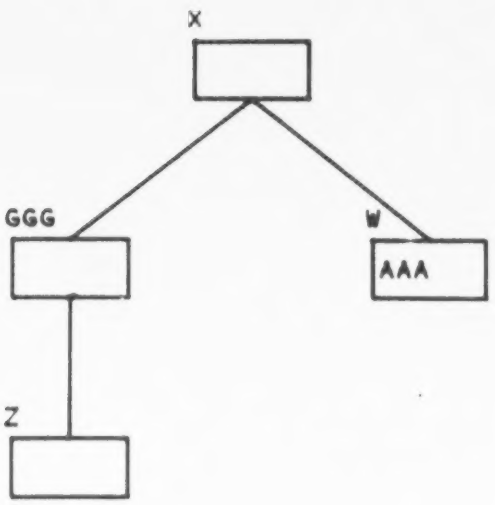
Figure 3



DISPLAY T02 RIGHT

Figure 4





DISPLAY BOTTOM

Figure 5

31

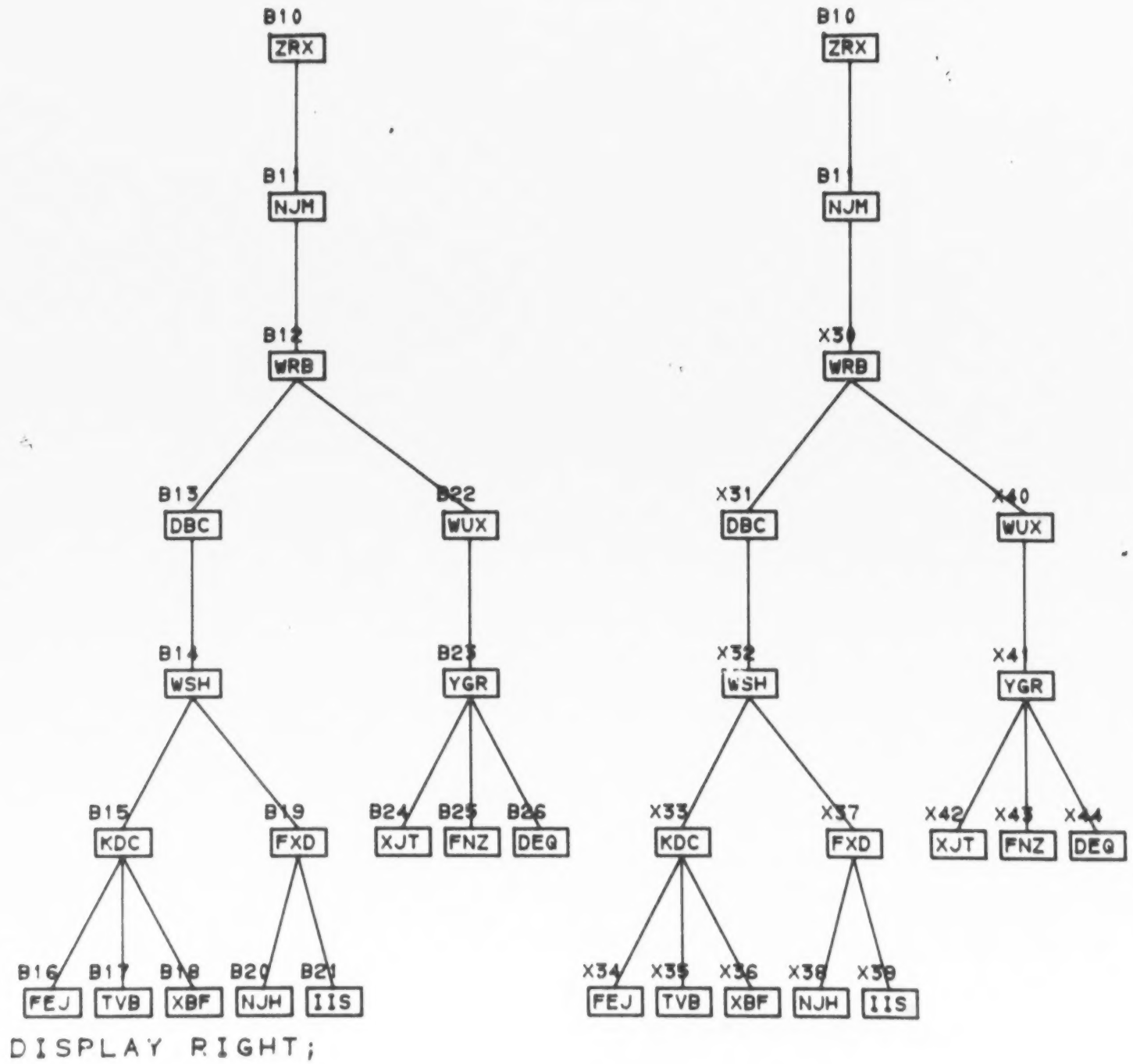
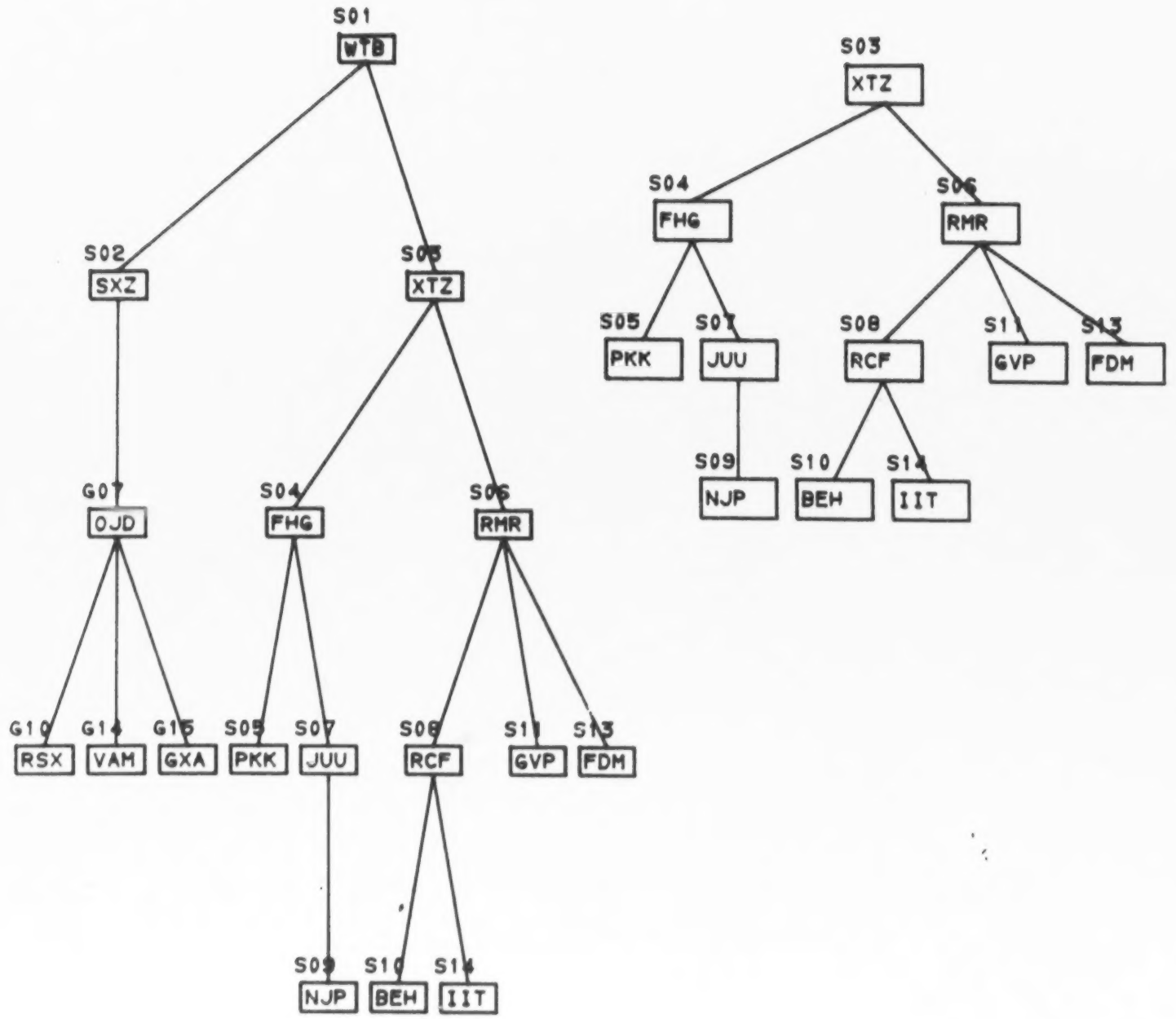

B10
ZRX
B11
NJM
B12
WRB
B13
DBC
B22
WUX
B14
WSH
B23
YGR
B15
KDC
B19
FXD
B24
XJT
B25
FNZ
B26
DEQ
B16
FEJ
B17
TVB
B18
XBF
B20
NJH
B21
IIS
X30
WRB
X31
DBC
X40
WUX
X32
WSH
X41
YGR
X33
KDC
X37
FXD
X42
XJT
X43
FNZ
X44
DEQ
X34
FEJ
X35
TVB
X36
XBF
X38
NJH
X39
IIS
DISPLAY RIGHT;

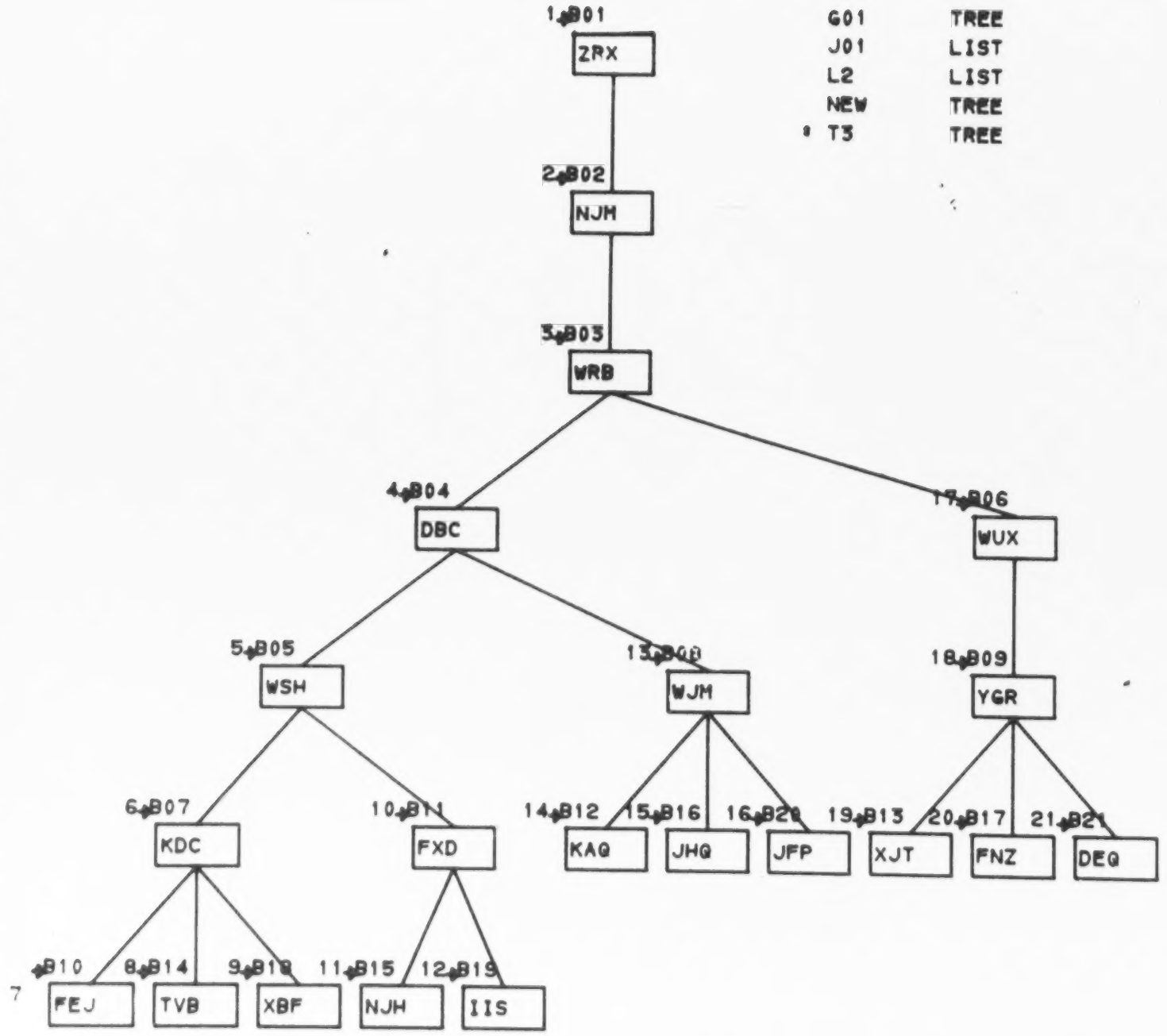
Figure 6




```
DISPLAY XXX TOP RIGHT;
```

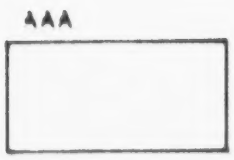
Figure 7

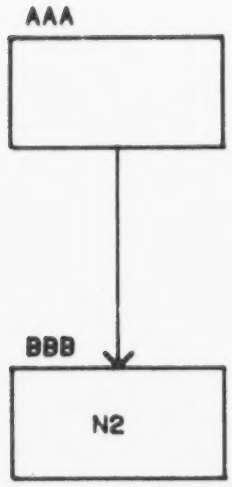
33

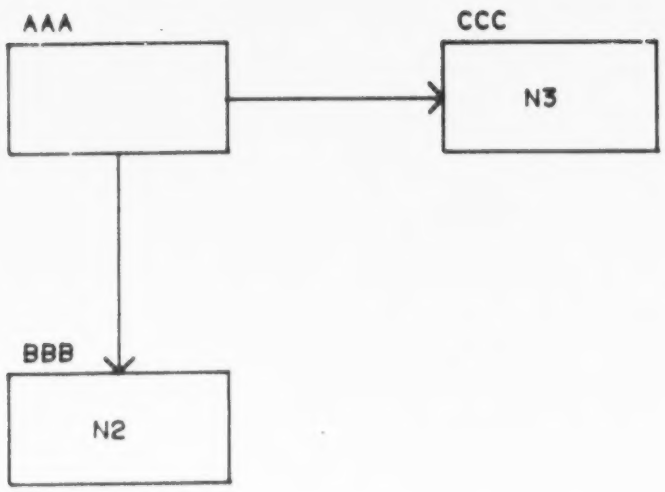


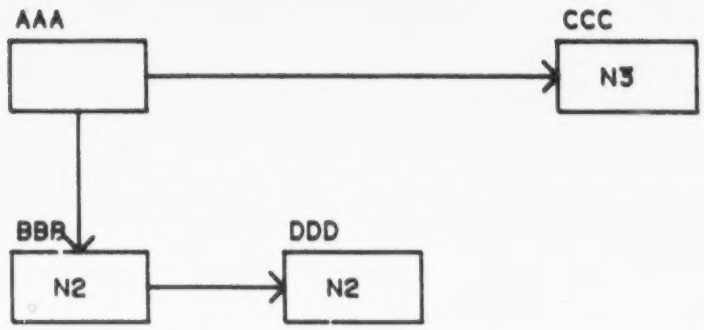
	G01	TREE
	J01	LIST
	L2	LIST
	NEW	TREE
*	T3	TREE

Figure 8



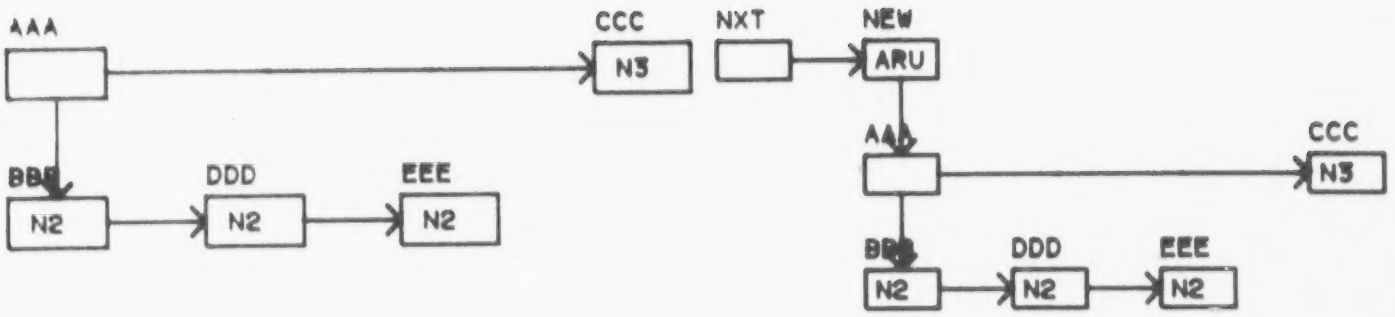


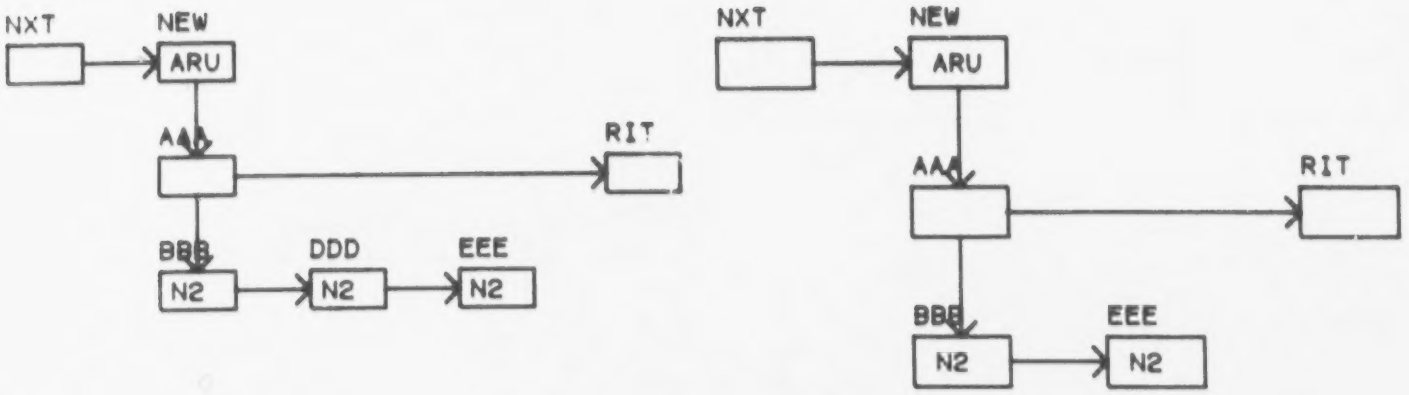





```
$DISPLAY BOTTOM RIGHT
```

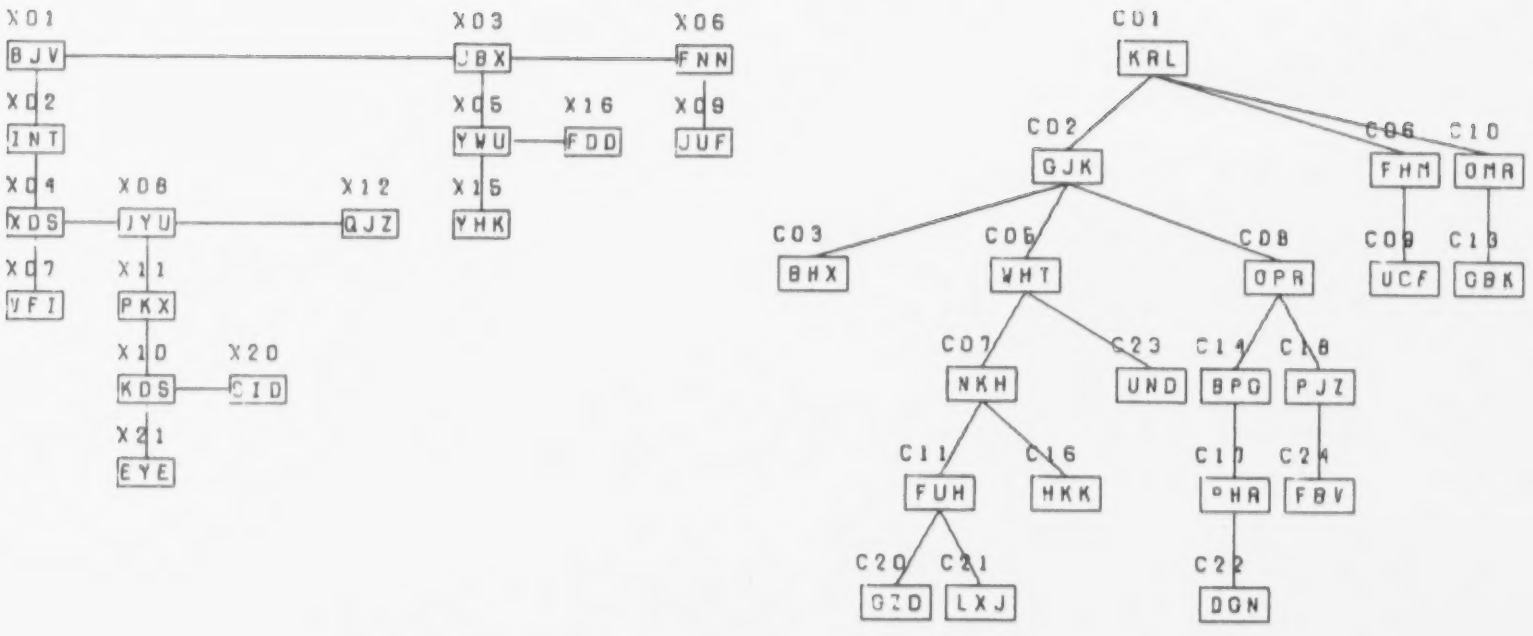
Figure 9

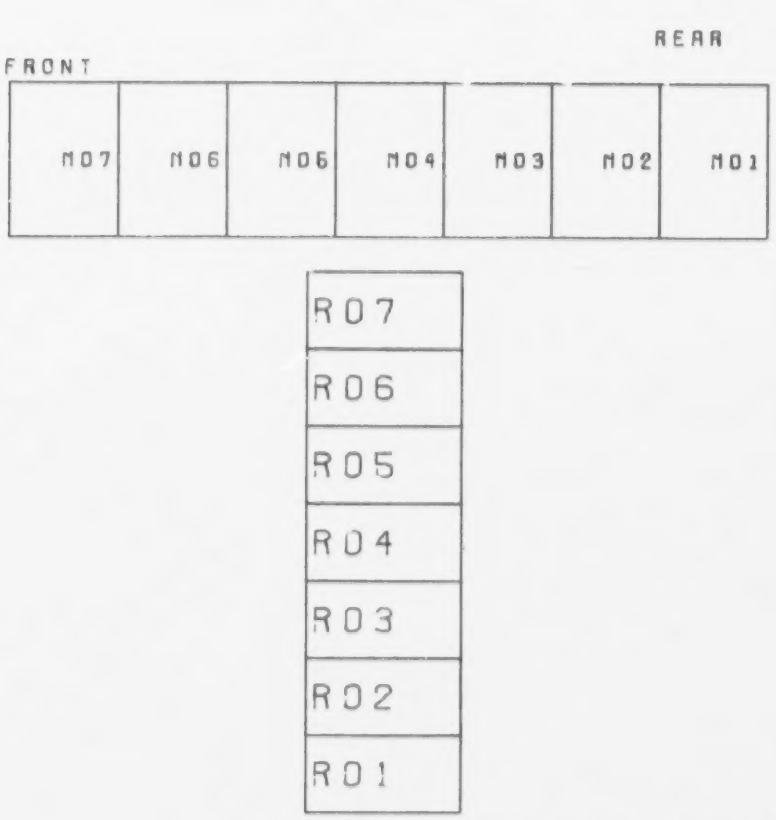




*DISPLAY BOTTOM RIGHT

Figure 10





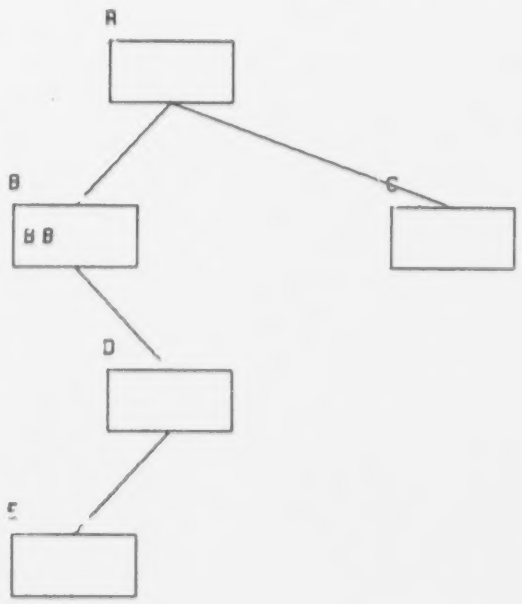


Figure 11. LIST,TREE,QUEUE,STACK and BTREE